Data Clustering by Large-Scale Adaptive Agent Systems

Ogston, E.F.Y.L.; van Steen, M.R.; Brazier, F.M.; Overeinder, B.J.

2005

document version
Publisher's PDF, also known as Version of record

Data Clustering by Large-Scale Adaptive Agent Systems

E. Ogston, M. van Steen, F. Brazier and B. Overeinder

Vrije Universiteit Amsterdam Technical Report IR-CS-014

Abstract

Finding items with specific characteristics in large distributed systems can be problematic. Directories, the most common way of enabling capability-based search in distributed systems, gather in a central location data which is often inherently decentralized. This paper considers an alternative decentralized method of enabling search, which leaves data objects in place at their natural location within a physically distributed network. This form of decentralized search is likely to gain importance as computer systems become more widely distributed and the autonomy of their components increases. Autonomy plays a key role in this development since it precludes the global use of a single comparison method for determining the similarity of objects. With this in mind, this work explores the ability of explicitly autonomous peer-to-peer agents to form themselves into groups within an overlay network, based on their locally perceived similarity, thus providing a structure that can facilitate search. We introduce a decentralized clustering procedure, designed to discover small clusters (of 500 items or less) in very large, widely distributed sets of data. This procedure is shown to scale well in the number of clusters. The paper further demonstrates that for 2D spatial data it produces clusterings that compare well to those produced by central clustering algorithms, especially for data sets that contain widely varying cluster characteristics.

1 Introduction

The field of multi-agent systems attempts to reduce the complexity involved in building large distributed systems by dividing functionality among small independent processing units, called agents. In theory these agents can be designed separately and later dynamically combined to solve problems that reach beyond their individual scope. In order to work together, however, agents need to first be able to find potential partners. The common solution to this, maintaining a central directory, can be expensive and runs counter to the concepts of autonomy and decentralization that agents embody. If, instead, agents were able to organize themselves into groups based on functionality, interests, or goals, the search problem could be greatly simplified [12] [7].

A similar problem arises when considering large volumes of data, created by many sources, in which a user would like to find items with particular characteristics. One method of enabling this search is to cluster the data into groups of like items. However, current

clustering algorithms assume a centralized view of data from which certain general characteristics can be determined. Thus, if an unknown data set is widely distributed over a network it must first be gathered before it can be clustered. This gathering task can become a costly operation. In this case a decentralized clustering algorithm that allows the data to remain in place, could be advantageous.

In this work we study decentralized clustering, motivated by the need to provide alternatives to centralized directory services for autonomous entities. Our aim is to group data items that are distributed across a geographically dispersed network, such as the World Wide Web, without knowing categories, or their characteristics, a-priori and without collecting the data items to a central location.

We create an adaptive overlay network that structures data into clusters that correspond to discovered categories. In this network each data item to be clustered is represented by an agent node. These agents interact in a peer-to-peer manner. An agent wraps together a data item with matching functionality. This matching functionality is used to independently determine the similarity of the agent's data item to the data items held by its neighbor agents in the peer-to-peer network. Based on item similarities, agents form themselves into groups, or clusters.

This model of clustering may seem a roundabout approach to a problem that already has many well-studied solutions. However, we believe that for use in autonomous distributed systems existing clustering algorithms are lacking in several important aspects. Traditional clustering algorithms are (conceptually) centralized, making the assumptions that all data, or a summary thereof, can be gathered into a single place, and that all data items can be compared in the same way. These assumptions do not always hold in decentralized systems, and especially in autonomous agent systems. In these systems a problem arises from the fact that the more flexibility we wish to have in comparing data items, the more information a central location needs to store. The more information that must be stored and transported, the higher the cost of maintaining the central location becomes. In the case where data items are fully autonomous entities, the information required to make comparisons could amount to the entire agent. Conceptually-centralized clustering algorithms and directories thus run into the following problems which can be more easily addressed in a decentralized model of clustering:

- *A central component may be too expensive to build.*

  Central services are expensive in terms of memory, processing, and communication resources. While they are usually technically possible to build, if a node in the system does not already possess the required resources they might not be practical to build. Moreover, if a central service does not already exist in a system spread over multiple management domains, an agreement must be reached on the location and protocols for that service. Again, this is a problem that can technically be solved, but the agreement protocol could be prohibitively expensive. For these reasons we explore a peer-to-peer system without additional supporting services.

- *A single definition of similarity may not cover all clusters in the data set.*

  Some data sets cannot be easily described in terms of a single global parameter. For instance if we were to define a city as a connected area with a population density of more then x people per kilometer, cities in the US would be identified properly, but the whole of Japan might be considered one city, and Australia might have no cities at all. In this case we need to associate with each cluster the similarity function that should be used to define it. If this is taken to the extreme, each data point might need to have its own similarity function, which brings us to the agent model where an agent encapsulates both data and matching policy.

- *A central clusterer may lack information it needs to determine matches.*

  If matching policies are not based on a standard set of characteristics a central component may not contain the information needed to decide if two items are similar to each other. If items match on known characteristics but precise similarity can only be determined based on further information, additional details can be requested. However, if two items appear not to match, but would do so considering their full characteristics a central component would never know that it should request more details. This false negative case is increasingly likely to occur among agents who might want to control access to private data or might not want to give a central clusterer the exact details of their matching function. Game theoretic agents for instance rely on the privacy of their preferences, like reservation price, in order to maximize their profits. By allowing agents to decide on matches locally we minimize this problem. Agents have all the information about their data points and can decide on the spot what details to share with whom.

- *The decision function required to determine individual matches may not be generalizable.*

  There are some preferences and characteristics that are impossible to express; the personality of an ideal job candidate, the feel and drape of a fabric, or the atmosphere of a city, for instance. Computer systems can have similar quirks, and often the only way to truly test them is to run them in context. By allowing agents to meet directly to determine compatibility we enable far more intricate decision making then can be realized through a third party.

In the following sections we develop and test an agent algorithm designed to address these issues. Through experimentation on abstract data sets we show that the time and resources required by each agent in this decentralized algorithm scales well with the number of clusters in a data set when clusters are limited in size, while producing clusterings of a quality that is comparable to that of more traditional algorithms. Moreover, we find that the flexibility in adapting matching criteria to local conditions provided by using agents allows our algorithm to discover clusters in data sets with widely varying cluster characteristics that cause difficulties for centralized algorithms.

These results expand on work in [9] and [10] which present a more basic form of decentralized clustering, based on a fixed cluster size, and explore its use in clustering text. In particular, this paper concentrates on the automatic derivation of values for design parameters used in our previous work, and present in various forms in other clustering algorithms. This paper provides an extended model of the agent algorithm and explores how agent adaption can be used to learn appropriate cluster boundaries. Extensive experimental results are provided showing how agents can learn cluster sizes, and comparing the clusterings produced by the resulting algorithm to those found by other clustering algorithms. We thus explore a manner in which the parameter space required to find clusters can be reduced.

Reducing the parameter space enables the use of clustering in decentralized systems. Such clusterings could be used to facilitate decentralized search by producing semantic groupings of data. By organizing distributed data in this manner the space that must be searched to answer a query can be reduced, as in semantic overlay networks [1] [13]. However, in current semantic overlays nodes individually determine a set of semantically close neighbors, while a clustering algorithm produces a set of explicit groups of nodes. These groups could possibly be used to improve the efficiency of the search process and quality of results by giving additional structure to the overlay network.

Section 2 formally describes the model we consider and Section 3 considers the main implementation issues. Section 4 explores this model's basic behavior through simulation experiments and adds some finer points to the implementation. Section 5 presents more detailed experiments on possible ways of allowing agents to learn about local conditions within a data set. Section 6 discusses related work, including a detailed explanation of why traditional clustering algorithms cannot be easily distributed. Finally, Section 7 provides conclusions and issues for future work.

2 Model Description

This section defines a model of decentralized data clustering in which items from a data set are each represented by simple agents. These agents implement individual matching policies for their data so that any pair of agents can independently reach a joint decision about the similarity of the two items they hold. The overall goal of this system is to have the agents, by means of decisions based on a severely limited view of the system, form themselves into groups such that the data held by the members of a group correspond to a globally correct cluster in the underlying data set. In order to coordinate group membership decisions, agents report a strength value for their autonomously chosen partnerships to their current group. The group as a whole compares the strengths of these proposed partnerships when admitting new group members or disbarring old ones.

More precisely, the problem is to organize a collection of N data objects $X = \{x_1, \ldots, x_N\}$ into nonoverlapping groups, or *clusters*, of similar objects. Each data object, x_i, is assumed to consist of a number of attributes, allowing us to represent complex data. To construct clusters, we consider a set of N agents $A = \{a_1, \ldots, a_N\}$, where agent $a_i \in A$ represents data

object $x_i \in X$.

Each agent, a_i, has a set of δ ports, $P_i = \{p_{i1}, \ldots, p_{i\delta}\}$. Agents communicate with other agents through their ports, allowing them to compare data objects. At each moment in time an agent has associated to each of its ports, $p_{im} \in P_i$, a communication *link* which joins it to another agent's port, $p_{jn} \in P_j$. A port should be thought of as a peephole that provides, to an outsider, a particular view on its agent's data. For complex data objects a port might allow access to only some subset of the full agent data, or might allow for one of only many possible forms of similarity comparisons.

A link, l, is made up of a pair of ports, $l = \langle p_{im}, p_{jn} \rangle$. For each of its links an agent, a_i, uses a *matching function*, $m_i : P_i \times P_j \rightarrow [0, \infty)$, to determine its *similarity* to the *neighbor* agent to which the link joins it. The higher the value of m_i, the more agent a_i believes that its data object and that of agent a_j are similar, based on the views available through the link's ports, p_{im} and p_{jn}.

Agents, however, do not necessarily agree on their similarity to each other. That is, it is possible that $m_i(l) \neq m_j(l)$. In order to simplify the comparison between links, we introduce a function, $s(l) = f(m_i(l), m_j(l))$, through which two agents can agree on a single *strength* for a link. This strength function could be, for instance, the maximum, minimum, or average of the two values, or for more advanced agents, could represent a negotiation process.

Links can be of three types depending on the relationship between the agents they join:

- **Nonmatching**: Links between agents with a low similarity. These links have a strength below a threshold value, σ_{low}, which is determined by the agents adjacent to the link.

- **Matching**: Matching links, or *matches*, are between similar agents, in the opinion of the adjacent agents. They have a strength between σ_{low} and a threshold variable determined by their cluster, σ_{high}.

- **Connecting**: Connecting links, or *connections*, are links between similar agents, in the opinion of those agents' cluster. These links have a strength above σ_{high}.

Note that σ_{low} and σ_{high} are not system parameters but values that are defined locally by the agents and clusters. In practice they may vary over time or be determined by probability functions instead of being fixed values.

Together the agents and links at a particular point in time, t, form an undirected graph G_t, with vertex set A and an edge set containing all the links. We define *agent clusters* by considering the graph G'_t with vertex set A and an edge set containing all the *connected* links at time t. Each connected component in G'_t is considered to form an agent cluster. Figure 1 shows a two-agent cluster.

Agents, or clusters of agents working together, are able to perform the following local actions in order to change the edge sets of G and G':

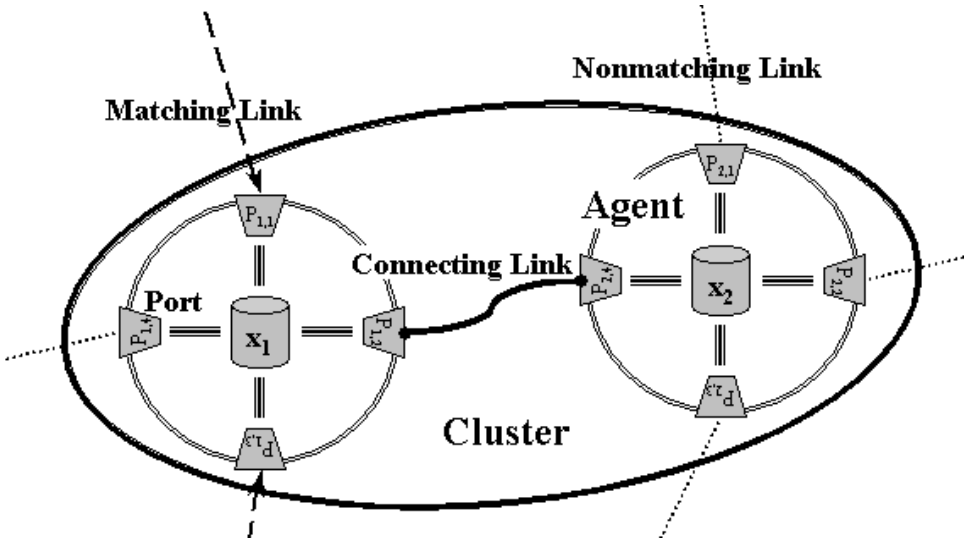


Figure 1: Diagram of two clustered agents, each with four ports.

- **Swapping nonmatching links**: Given two nonmatching links, $l_1 = \langle p_{cm}, p_{dn} \rangle$ and $l_2 = \langle p_{eq}, p_{fr} \rangle$, if agents a_c and a_e are in the same cluster, they are able to swap their links, producing $l_1' = \langle p_{eq}, p_{dn} \rangle$ and $l_2' = \langle p_{cm}, p_{fr} \rangle$. This action performed repeatedly among agents in a cluster results in a permutation of the cluster's nonmatching link set.

- **Upgrading nonmatching links**: *Agents* can use their matching functions to evaluate nonmatching links. If the resulting strength is above their current joint threshold σ_{low}, the two agents adjacent to the link can upgrade a nonmatching link to a matching link.

- **Upgrading matching links**: *Clusters* can use their current threshold σ_{high} to choose a matching link to be upgraded to a connecting link. This can result in two clusters being merged into a single cluster.

- **Breaking matching and connecting links**: *Clusters* can choose to downgrade a matching or a connecting link to a nonmatching link. This action can result in a cluster splitting into two smaller clusters.

We add one further element to the model, a maximum cluster size, s, which defines the maximum number of agents a cluster can contain. When this limit is reached no further connections are made until the cluster size is again reduced by breaking a connecting link. This requirement does not have a functional purpose, but is essential in maintaining decentralization since it limits the cost of coordination within clusters.

Over time, as the agents perform the actions listed above, the graph G' changes. Our aim is for the series of graphs $G_0', G_1', \ldots G_t'$ to converge to a graph G_*' in which the agent clusters

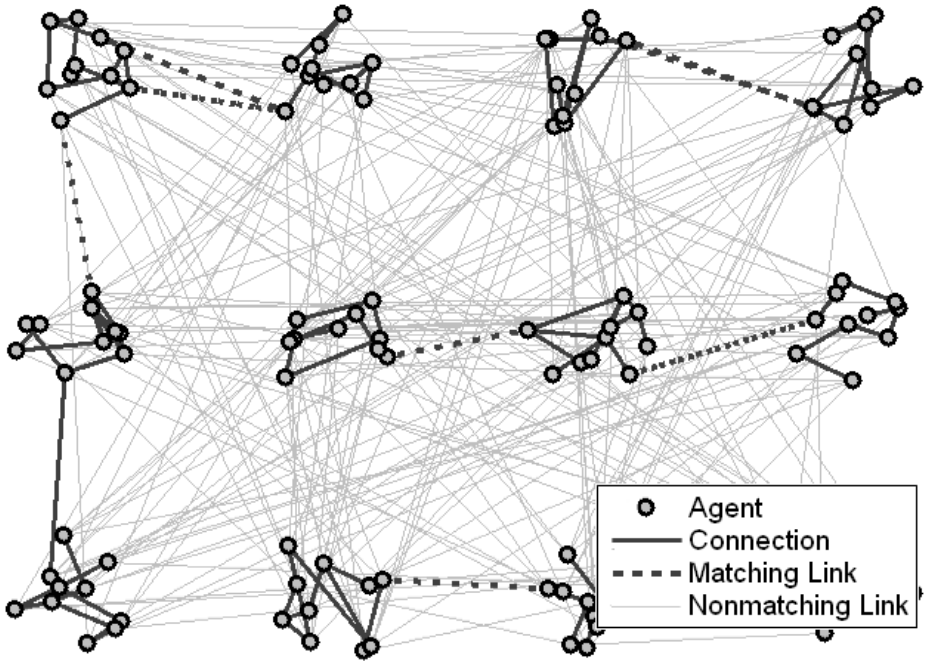


Figure 2: A set of agents, placed at their data object's coordinate, and the links between them during a typical experimental run.

correspond to the expected clusters produced by a clustering algorithm working with the *data space*, X. In the experiments in this paper we will adjust the model's behavior by modifying the *agent decision functions*: the matching functions by which agents choose to upgrade nonmatching links, and the functions by which clusters choose to upgrade matching links or break existing links. We will picture and analyze the clusters found in data space, allowing us to compare the effects of the agent clustering with traditional data clustering algorithms. Figure 2 shows a set of 120 agents representing 2-D spatial points, and the links between them at a particular time during a sample run. Each agent is drawn at its coordinates in the data space, while the links show the positions of the agents in *agent space*.

The agent clustering procedure defined above is designed to allow for a wide range of flexibility in the kinds of data that can be clustered. To provide implementations for different types of data one simply needs to fill in the agent functions that depend on the data type, that is, the matching functions. Since these matching functions return a strength, the type of which could also change with the data type, it may also be necessary to change the cluster functions that use this strength: the method of choosing new connecting links and the method of choosing links to break.

Defining these functions of course depends on the ability to find a strength metric common between all agents in a cluster. Agents determine matches independently, but clusters

need to be able to compare links when choosing which connections to make and break. If agents measure strength using vastly different metrics this comparison would not be possible. In many applications, however, such a metric exists naturally, and thus we will assume that strengths determined by different agents can be compared with one and other.

The abstractness of the procedure further allows for complex data types. By creating different types of ports for the same agent complex comparisons can be reduced to a set of simpler assessments. For instance, data objects could be a set of documents written in a common language and link strengths could be based on a guess at the similarity of their subject, such as the angle between vectors of chosen keywords [10]. Data objects could alternatively represent joint tasks with link strengths based on the bandwidth available between two agents working on a task together. Data objects could even be the users agents represent, while link strength could be based on the amount of time each day that their PDA's spend in the same room, or the amount of money users are willing to pay to maintain a link.

On the other hand, because the "correct" clustering of a set of data is impossible to define, due to subjectivity, there are no satisfactory metrics for evaluating algorithms that cluster complex data types. In this paper our aim is to investigate how clustering behavior changes as agent behavior is modified. Thus, the experiments presented use simple two-dimensional spatial data points, with the Euclidean distance between them as a similarity metric. This is a domain in which clustering has traditionally been studied, and in which the clusterings produced are relatively easy to analyze. This approach allows us to assess the quality of the decentralized agent method in comparison to traditional nondistributed clustering methods.

3 From Model to Implementation

Figure 3 gives a high-level pseudo-code specification of the algorithm underlying the model in Section 2. In this section we first describe the pseudo-language used to describe agent operations, and then detail how the agents function.

The code depicts agents and clusters as distributed objects that communicate through remote procedure calls (RPC). The objects have the following components:

- **Process**: Object operations are carried out by a number of concurrent processes. Process methods are started when an object is initialized. Arguments in square brackets indicate that a number of processes are created, one for each argument.
- **External procedure**: An external procedure can be invoked through an RPC from outside the object. These procedures operate concurrently: each time a call is made a new process is started to run the procedure.
- **Internal procedure**: An internal procedure is available only from within the object itself.

```
1.  object Agent{
2.    acquaintances: set of [ neighbor:Agent, matching:Boolean, connecting:Boolean ]; //Agent's view of its adjacent links.
3.    c: Clique; //The clique of which this agent is a member
4.    process searchForMatches [for each aq in acquaintances]{ //One process for each acquaintance
5.     whenever ( ''now and then'' & matching = FALSE & connecting = FALSE ){
6.       next:Agent := c.tradeIn( aq.neighbor );
7.       aq:= [ next, ''negotiate match with next'', FALSE ];
8.     }
9.   }
10.  virtual object Clique {
11.    members: set of Agent; //All agents that are members of this clique
12.    unwantedAcquaintances: set of Agent;
13.    external procedure tradeIn( a:Agent ) returns ( a':Agent ){
14.      unwantedAcquaintances.add( a );
15.      wait until( ''sufficient unwanted acquaintances have been received'' );
16.      return unwantedAcquaintances.remove( ''choose an unwanted acquaintance to return'' );
17.    }
18.    process searchForConnections{
19.     whenever (''good matches have been found'' ){
20.       nextLink:[ neighbor:Agent, matching:Boolean, FALSE] = ''choose an unconnected link'';
22.       if ( members.contains( nextLink.neighbor ) ){
23.         ''upgrade nextLink into a connection'';
24.       }else if ( bestMatch.neighbor.c.requestConnection( self ) = ACCEPT ){
25.         ''upgrade nextLink into a connection'';
26.       }
27.     }
28.   }
29.    external procedure requestConnection(requester:Clique) returns (ACCEPT, REFUSE) when (''available''){
30.     if (''proposed clique OK''){
31.       return ACCEPT;
32.     }else return REFUSE;
33.   }
34.    process reconsiderCompositon(){
35.     whenever (''current composition may be improvable''){
36.       nextMatch:[ neighbor:Agent, matching:FALSE, connecting:Boolean ] = ''choose a matching link'';
37.       nextConn:[ neighbor:Agent, matching:Boolean, connecting:FALSE ] = ''choose a connecting link'';
38.       if ( nextMatch != null ){ ''downgrade nextMatch to a nonmatching link''; }
39.       if ( nextConn != null ){
40.         ''downgrade nextConn to a nonConnecting link'';
41.         if (''clique is no longer connected''){''split off a new clique'';}
42.       }
43.     }
44.   }
45. }
```

Figure 3: High level agent and clique implementation.

- **Variables**: Variables are local data, visible only within an object. We follow the convention that data is implicity protected from concurrent access.

The agent paradigm has some peculiarities beyond the standard distributed programming conventions. First, an emphasis is placed on the fact that agent actions can be initiated either by an external request or by some internal trigger. We indicate this idea of "autonomous action" by means of the **whenever** keyword which states that a process starts a given action each time the given condition is met. Furthermore, agents are considered free to choose whether or not to act upon external requests. We indicate this by means of the **when** keyword which gives a condition for acceptance of requests. Finally, in a full agent model calls can be lost, or even misinterpreted. For simplicity's sake our implementation assumes perfect messaging with a notification of ignored calls, though in our design we do keep the possibility of lost calls in mind.

The pseudo-code describes two types of system objects, agents and clusters. Clusters, however, exist only in concept, not as actual entities. This fact is indicated by the **virtual** tag. Cluster operations must in fact be achieved through cooperation among a number of distributed member agents. Some actions require information about a cluster's members as a whole, as indicated by the set notation *.all* and *.contains*. Others, like upgrading and downgrading links, require informing members affected by the action. In consequence, the actual implementation of a cluster involves a fairly large amount of communication between its members. This communication load becomes a limiting factor in the number of agents a cluster may contain, reflected by the limit, *s*, placed on the cluster size. Our actual implementation of cluster operations uses a simple coordination mechanism; we specify that in each cluster one agent is elected to act as the "head" of the cluster and maintains the information needed to carry out joint operations.

Coordination between clusters, on the other hand, is not done explicitly. In places, agents or clusters would like to know that a certain global condition has been met, yet as local entities have no means of checking. The code's design instead works on the principle that over time certain assertions are likely to become true. This implies that agents merely have to wait for a time, after which they can simply start to act. The underlying assumption is that by the time they take actions the desired conditions will prevail. If by some chance the conditions did not hold and a mistake is made the algorithm is designed in such a way that corrective actions are naturally taken. In other words, from a correctness point of view no harm is done by taking premature actions, they will only affect performance.

With the above conventions in mind, the details of the agent clustering operation can be explained as follows. Agents each have a set of acquaintances (line 2) which represent their individual view of links to other agents. Agents further have a process, *searchForMatches* (lines 4-8), which is run once for each acquaintance, as indicated by its argument. This process implements the swapping of unmatched links with other cluster members and determines if new links should become matches. The process's **whenever** statement (line 5) continually checks an acquaintance's status. When this status becomes "nonmatching", a request for a new neighbor is made to the agent's cluster, by means of the cluster's *tradeIn*

procedure (line 6). When *tradeIn* returns, the acquaintance is reset to form a new link, calling procedures in the new neighbor agent to determine a status and a strength (line 7). Note that *tradeIn* (lines 13-17) is an external procedure, meaning that each invocation is handled by a separate process.

The clusters play a coordinating role in this routine through the *tradeIn* procedure. This procedure collects unwanted link ends, shuffles them in some way and then returns them to waiting agents. In the model this is described as a permutation of the link ends. A true permutation could be achieved if "sufficient" in line 15 was taken to mean all nonmatching links in the cluster. This condition synchronizes the speed at which cluster agents search and synchronizes a cluster's actions with its neighbors, yet also forces the cluster to act at the speed of its slowest member or neighbor. In our experiments, this requirement is relaxed so that unwanted acquaintances are returned each time 40%, rather then 100%, of currently nonmatching links in the cluster have been traded in. Over time the mixing behavior results in agents finding possible matching links among their cluster's neighbor agents.

The aim of the clusters is to choose the best among the matching links found by the agents to form connections, thus mutating the cluster composition. This is done by the process *serachForConnections* (lines 18-30) which creates new connections, and the procedure *reconsiderComposition* (lines 36-43) which picks unprofitable matches and connections to downgrade. These procedures act on the principle that all connections that can be made should be made, until the cluster's limits have been reached at which point steps should be taken to ensure that cluster assets are used optimally. Thus when better matches are found that cannot be made, bad connections should be broken to make room in the cluster while all other matches are unneeded and should be discarded to free links to continue searching for better ones.

The *searchForConnections* process is governed by a **whenever** clause in line 19 which pauses operation until a "sufficient" number of matches have been found. The purpose here is to synchronize with the rate at which agents search for matches, approximating the underlying assumption that the matches considered are the best currently available among the cluster's neighbors. If new connections are chosen too quickly they will only have to be broken again, which is a fairly costly process since it involves many changes to the cluster composition. In practice we implement "sufficient" as "each time the trade in process returns new acquaintances". The additional "no composition changes pending" condition is used to ensure that a cluster can be involved in negotiating only one merge or split operation at a time. This simplifies the clause in *requestConnection* which determines if a connection request should be accepted (line 32) to, in this implementation, a straightforward check of the size of the resulting cluster should the connection be formed.

When a connection request fails both clusters involved call *reconsiderComposition* to check if they in fact make the best use of available links. In principle, the worst link will be downgraded. A deterministic choice can however, in practice, result in a race condition where a match is repeatedly upgraded and downgraded, protecting a slightly better link that could be broken to more profit. Thus the choice clauses in lines 40 and 42 are used to introduce some degree of randomness in choosing not the worst link but among all of the

not so good links.

Because of the weak synchronization mechanisms used among agents, simulating and measuring time is particularly difficult since it cannot be assumed that agents will all act at the same speed. Concrete agents could require vastly different amounts of computation time to decide matches, run on different speed processors, and bandwidth between agents could vary dramatically. In the simulations in this paper agents run at roughly the same speed. On the other hand, as we have just discussed, the timing of actions is not based on a common clock tick, but trigged by messages received. We thus measure time in *turns*, one turn consisting of $100 \times N$ messages, where N is the number of agents. Given the fact that all agents handle roughly the same number of messages, this provides an arbitrary time interval which allows us to compare the estimated running time of systems with varying numbers of agents. Since agent operations are simple, and limited in size by the maximum cluster size, this estimation reflects the assumptions that communication will be the largest cost in the system, and that agents act relatively independently of each other.

4 Experimentation

The first experimental task is to determine if the agent procedure described in the previous sections actually produces reasonable data clusters. To find data clusters agents need to locate and identify their near neighbors in data space. Since agents are linked to a limited number of neighbors in agent space it is quite possible that they will not be able to do this. It could happen that agents find no matches among their initial neighbors, resulting in no clusters forming. Likewise, agents may never discover their neighbors in data space, resulting in data clusters that are split in pieces or intermingled among the agent clusters, or agent clusters that spread too widely over the data clusters. In the following sections we explore basic agent clustering behavior by first studying agents with naive versions of the decision functions discussed in Section 3, and then gradually increasing decision function complexity to improve cluster quality.

4.1 Basic Behavior

In this section we illustrate the basic clustering ability of the agent procedure. To this end we consider a simple data set made up of two-dimensional real-value points in clearly separated clusters. The goal is for agents to discover these prearranged clusters.

Figure 4 shows the set of points considered, and the clusters found by their corresponding agents using a simple matching strategy, described shortly. The markers are positioned at each agent's location in data space. The links between agents are not shown, but the color and shape of a marker represent a point's agent cluster membership. The agents used for this example follow a primitive matching strategy: we simply set the strength of a link joining two agents to the Euclidean distance between them, and use a fixed strength threshold,

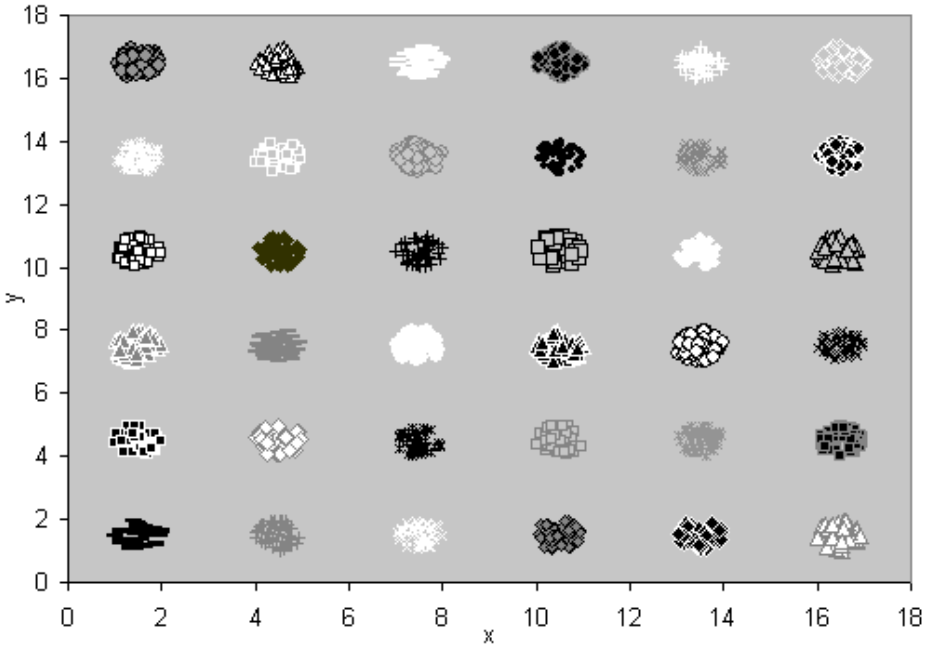


Figure 4: A simple data set (point location) and the agent clusters found (point coloring) using naive agents, $\lambda = 5$.

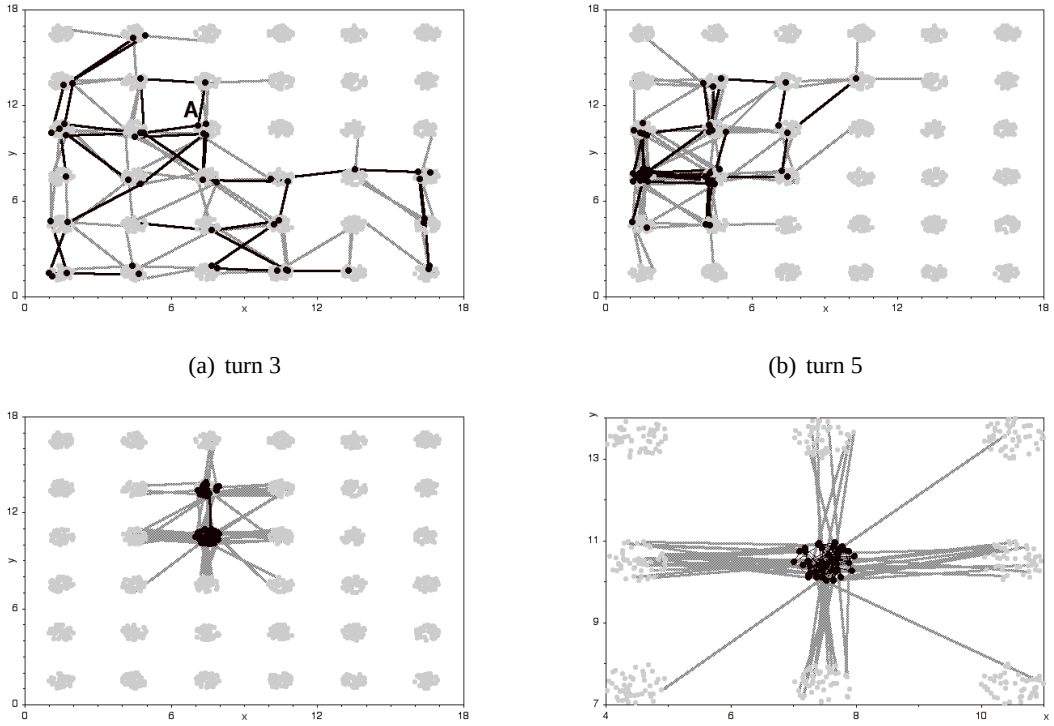
λ, to determine matches. The resulting matching function is given by the agent procedure:

```
external procedure negotiateMatch( other: Agent ) returns ( ACCEPT, REFUSE ){
  distance: Real := ''Euclidean distance between this agent's and other's data points'';
  if (''true with probability 1 − distance/λ '' & other != self ){
    return ACCEPT;
  }else return REFUSE;
}
```
(1)

It can be seen in Figure 4 that even these simple agents are quite effective; the agent clusters correspond precisely to the expected data clusters. On the other hand, the agents had some a-priori knowledge of the data set; they were given $\lambda = 5$, and a maximum cluster size, $s = 75$. These values represent the approximate density and size of the clusters to be found. Consequently, the task was not as difficult as it could be. It was not obvious, however, that the agents would find the data clusters correctly. The maximum length of a matching link was 5 units, large enough to span the space between two data clusters. The maximum cluster size was 50% larger than the actual data cluster size of 50. Thus randomly searching for matches and replacing connections with better ones as they are found, endows agents with a nontrivial ability to accurately discover boundaries between data clusters.

Figure 5 shows the development over time of a sample agent cluster, from the experimental run shown in Figure 4. For all of the experiments in this paper we use an initial configuration in which agents are randomly linked, and each agent forms a separate cluster. This setup approximates a wholly unorganized system. Thus at the start of a simulation agents choose partners for early clusters from among a very limited number of random neighbors, resulting in poor matches and connections. Figure 5(a) shows an early configuration in which the sample cluster contains 33 agents, represented by black dots. Other agents in the data set are represented by white dots. The agents in the sample cluster are more or less randomly scattered over the data set and have little correspondence to any actual data cluster. The solid lines indicate the sample cluster's connecting links, and the dashed lines indicate the matching links. These show that most agents have found neighbors in adjoining data clusters, but very few agents have found a neighbor within their own data clusters. Notice that breaking a single connecting link, at point A for instance, breaks the sample cluster into two better clusters without requiring any other changes.

Figure 5(b) shows the sample cluster a little further into the run. It now contains 65 agents, near the maximum cluster size of 75. At this stage the cluster is compacting and the search procedure has allowed many more of the agents to find connections to other agents within their own data cluster. Figure 5(c) shows the sample cluster near the end of the run. Here it contains 70 agents making it so large that it spans two data clusters. However, these two parts of the cluster are joined only by a single connecting link, which, due to its longer than average length, is likely to be broken and not remade. In the final image, Figure 5(d), the sample cluster has adjusted to the desired size of 50 agents. At this point all the agents have found links to other agents in the same data cluster. These links have been chosen as connections. The matching links and nonmatching links (not shown), continue to be used



(a) turn 3

(b) turn 5

(c) turn 15

(d) turn 21

Figure 5: The development of the cluster containing the agent (7.1, 10.7). Member agents are shown in black, other points in the data set are shown in grey. Connections are shown as black lines, and matching links as grey lines. Nonmatching links are not shown.

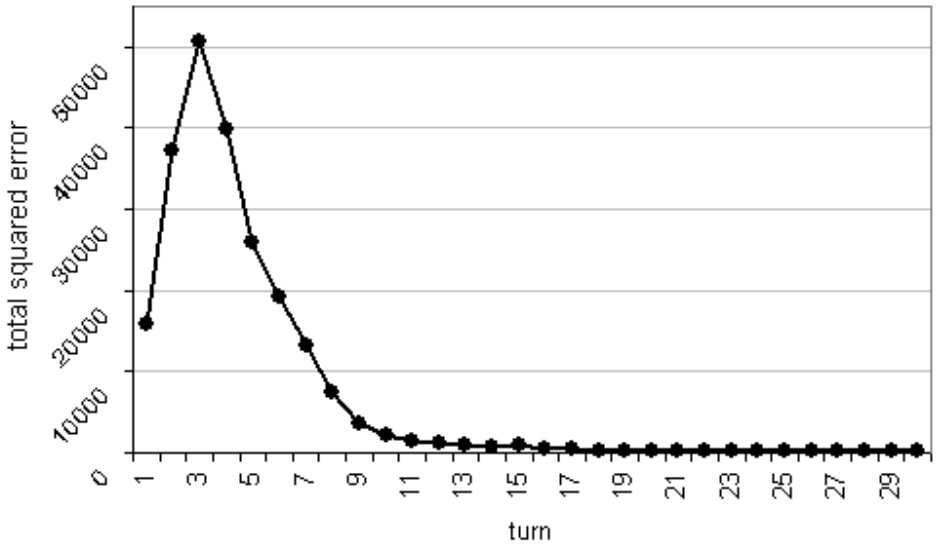


Figure 6: Total squared error over time for the experimental run in Figure 5.

to search for other better neighbors. This behavior is unneeded here, but allows for data sets which change over time. Once all clusters have also reached the correct size of 50 the cluster size limit of 75 prevents neighboring clusters from merging. Even without a cluster size limit connections made between data clusters will only be temporary since they will be much longer than connections within the data clusters. The cluster size limit is, however, required to stop interim clusters growing too large, especially during the early expansion phase.

In order to measure how quickly the structure of the agent clusters converges to that of the data clusters, Figure 6 shows, over time, a measure of the quality of the total clustering of the data set: total squared error, E^2. Given k clusters $C_1, \ldots, C_k$, where C_i has the Euclidean center m_i for $1 \leq i \leq k$, E^2 is defined as

$$E^2 = \sum_{i=1}^{k} \sum_{x \in C_i} \|x - m_i\|^2.$$

E^2 measures the compactness of clusters around a central point. While the E^2 measure has some weaknesses, for instance it favors clusterings with many small clusters, it gives us an adequate estimation of the quality of initial agent clusterings.

In Figure 6 E^2 is initially 0, since the original single agent clusters each have maximum compactness. As agents join together into the preliminary loose clusters E^2 climbs dramatically. At turn four it is close to the value expected from a random clustering. After turn four, however, the agents find increasingly better cluster partners and the clustering is

quickly improved. By turn ten E^2 drops to 5% of its maximum value, corresponding to relatively good clusters. From there the last improvements to the clusters, correctly placing the last few agents, are found more slowly, until E^2 reaches the value corresponding to the correct clustering in turn sixteen. Though finding the perfect clustering takes more time, the rapid convergence to tolerably good clusters is promising behavior. Section 5.3 discusses the convergence behavior in more detail.

4.2 Removing A-Priori Knowledge

When considering a wider range of data sets several problems with the algorithm described in Section 4.1 are encountered. The agents require two pieces of data-dependent knowledge: λ, the approximate maximum length of a good link, and s, the approximate maximum cluster size. It is important to remove this need for a-priori information since it is often difficult to determine the correct values of such parameters when a data set is unknown and distributed. In the following sections we discuss how these parameters could be determined during the clustering procedure.

4.2.1 Learning the matching range

In version 1 of the agent matching procedure, given in Equation 1, a nonmatching link was potentially updated if the distance between the two associated agents was shorter then a predefined maximum λ. Determining an appropriate value for λ is not entirely straightforward. A λ smaller than the minimum distance between two clusters results in only intra-cluster links becoming matches. While this ensures that only good clusters can be produced, it limits an agent's ability to join clusters when no good partners are available. Experiments are initiated by creating random links between agents. For small λ, the initial probability that a link joins two potentially matching agents becomes so small that the system remains a set of unconnected agents. On the other hand, if λ is increased to raise the probability of finding initial matches, the problem arises that too many not-so-good links are accepted as matches, leaving no ports free to search for better ones. This results in a very slow cluster improvement phase. A more desirable agent behavior would try to find matches near distance 0, but failing this, would over time relax demands and consider weaker and weaker matches. While the probabilistic matching function used in Equation 1 approximates this behavior to some extent, a better solution would have agents learn the correct value of λ for their current environment.

This learning can be achieved if agents observe the possible links that are presented to their ports and use this series to adjust their λ values. In the matching function shown in Equation 2, λ is initially set to 0 and then updated each time *negotiateMatch* is called. The procedure *updateLambda* increases λ by watching a series of distances, then slowly increases λ to the smallest values seen. When a match is found, λ is decreased to the

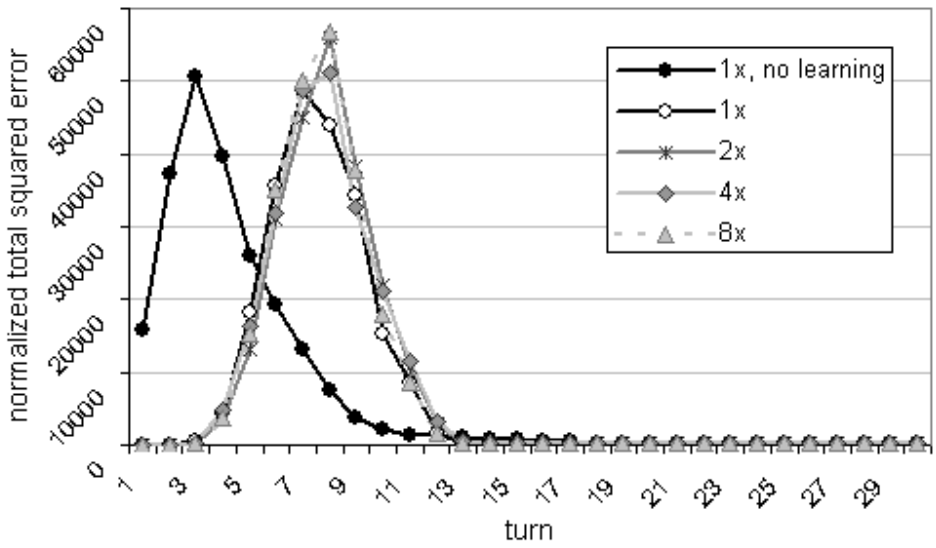


Figure 7: Normalized Total Squared Error over time for learning agents and stretched data sets.

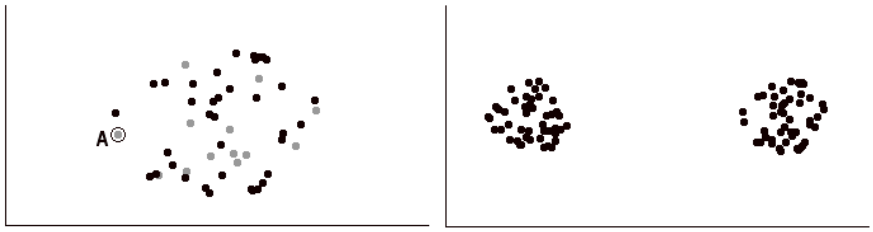
distance of that matching link.

```
λ: Real := 0;
external procedure negotiateMatch( other: Agent ) returns ( ACCEPT, REFUSE ){
  distance: Real := ''Euclidean distance between this agent's and other's data points'';
  updateLambda( distance ); //Possibly increase λ
  if (''true with probability 1 − distance/λ '' & other != self ){
    λ := distance; //Decrease λ so that next match found will be an improvement
    return ACCEPT;
  }else return REFUSE;
}

target: Real; step: Real :=0; count: Integer := 0;
internal procedure updateLambda( distance: Real ){
  if ( distance < λ ){ count := 0; step := 0 } //λ is large enough
  if ( count++ < 50 & step = 0 ){ target := minimum( target, distance ); } //Watch a series of distances
  else if ( step = 0 ){ step = (target − λ)/100; } //Slowly increase lambda to the smallest distances seen
  else if( λ < step ){ λ = λ + step; }
  else { step := 0; count := 0; } //Restart loop
}
```

(2)

Figure 7 presents the resulting E^2 graph for agents with this added component of behavior. It shows curves for the data set in Figure 4 with a fixed $\lambda = 5$, as before, and for the new learning agents. Since λ essentially defines the expected density of clusters, the graph also shows curves for the same data set stretched in both the *x* and *y* directions by factors of 2, 4 and 8. To account for the difference in distance between agents in these clusters, these curves were normalized by dividing E^2 by 2^2, 4^2, and 8^2 respectively. As can be expected, the learning agents take longer to converge to the correct clustering since they require some



(a) (b)

Figure 8: An example of a good agent cluster (a) versus an agent cluster that covers two data clusters (b). Black points are in the agent cluster, grey points are unfound points in the data cluster. The circle in (a), near label A, marks the current nearest matched point.

time to adjust λ. On the other hand, the fact that all the curves converge to the same value shows that in all cases the correct clustering was found. This implies that the agents discover λ correctly. Thus, by adding a simple learning procedure, it is possible to remove the need to define a-priori the scale at which clusters are to be found.

4.2.2 Learning the maximum cluster size

The second piece of knowledge required by the naive agents in section 4.1 was a maximum size, *s*, to which clusters can grow. The actual size of clusters, like the density of clusters, is data set dependent. Thus, in addition to learning λ, agents need to also learn the correct value of *s*.

The correct cluster size, however, is more difficult for agents to determine by observing only their surroundings. Let us hypothesize that clusters have a function, *shouldSplit* that returns true if they are made up of more than one data cluster, and false otherwise. Adjusting *s* then becomes straightforward. A cluster first grows to some initial *s*. It then waits until its membership becomes stable. Figure 8 shows us two examples of possible configurations of the cluster at this point. In Figure 8(a) the agent cluster contains points from a single data cluster and *shouldSplit* will return false. In Figure 8(b) the agent cluster covers points in two separate data clusters and *shouldSplit* will return true.

Thus once a cluster's membership is stable, if *shouldSplit* is true an agent cluster knows that it has grown too large and should break into two, resetting *s* for each of the resulting clusters. These clusters can then wait until their composition stabilizes and repeat the process to provide further adjustments. On the other hand, if *shouldSplit* returns false the cluster knows only that it is either the correct size or too small, and thus needs to check if it should grow. To do this it can simply run *shouldSplit* again, considering its nearest matched point to also be part of the cluster. In the example in Figure 8(a) the nearest matched point is marked with a circle, near label A. If *shouldSplit* still returns false the cluster knows there

is at least one more point that it should include and thus it should increase *s*. A call to the resulting cluster function, *reconsiderSize*, can be inserted in the *reconsiderComposition* function in Figure 3 before line 37.

```
type Link {a1: Agent; a2: Agent; strength: Real} //Helper type that defines a cluster's view of a link
internal procedure reconsiderSize(){
  if ( ! isStable() ) return; //Wait until the cluster composition stabalizes
  unwantedLinks: set of Link := shouldSplit(); //Returns links that should be broken to correctly split the cluster
  if ( unwantedLinks.size() > 0 ){; //The cluster is too large
    ''downgrade all unwantedLinks to nonmatching'';
    while ''the cluster members are connected'' { ''split off a new cluster''; }
  }else { //Check if next connection to be made would also fit in the cluster
    ''temporarily add the agent that the best matching link connects to to the cluster'';
    if shouldSplit().size() = 0 { ''increase s''; } //If the resulting cluster is still good increase s
    ''remove the temporary addition'' ;
  }
}
```

(3)

The growth process achieved through *reconsiderSize* is synchronized by the *isStable* condition. A cluster could determine if its composition has stabilized by recording and analyzing its membership over time. We opt, however, for the simpler method used in the other agent synchronization procedures: just wait for a long time. Again, mistakenly growing a cluster is not a problem in terms of correctness since the cluster will simply split at a later time. This does, however, effect performance since clusters that are allowed to grow too quickly they can engulf several data clusters before being correctly split.

Figure 9 shows the clusters found by these improved agents for a data set containing clusters of varying density, ranging from 20 to 200 points in size, with *s* initially set to 10. These agents were given a "perfect" *shouldSplit* function which used the knowledge that the minimum distance between two clusters in this data set is 2.0 units, as shown in Equation 4:

```
minInterClusterDistance: Real = 2.0;
internal procedure shouldSplit() returns set of Link{ //Returns connections that should be broken to split this cluster correctly
  unwantedLinks: set of Link := ''all connections for which strength ≥ minInterClusterDistance'';
  return unwantedLinks;
}
```

(4)

Figure 9 shows that though they no longer find perfect clusters, these new agents are now able to learn the correct size of the data clusters. There are a couple of typical mistakes. At point A two data clusters have been combined. This has occurred because one data cluster is much larger then the other. When clusters split the new *s* values were reset to 1.5 times their cluster's size, giving the new clusters some room to grow should they be missing points. This extra room makes a large difference to the speed at which clusters form, but allows very small clusters to be repeatedly added to, and then again split off from

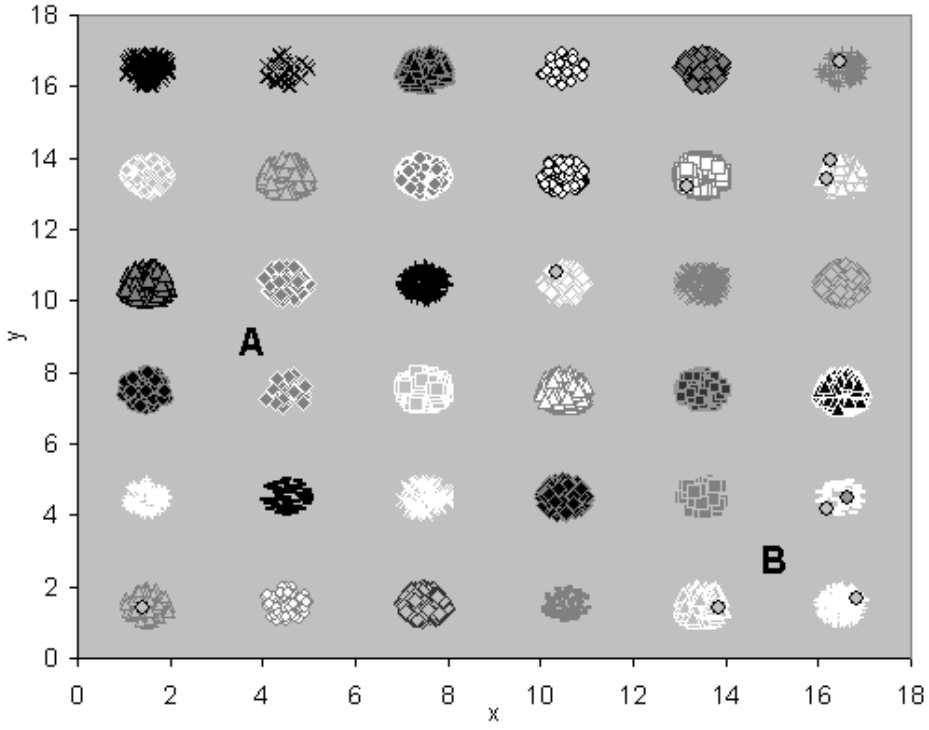


Figure 9: Learning agents' clustering of a data set with clusters of varying density. Grey circles are single agent clusters.

neighboring large ones. Figure 9 also indicates, as at point B, that some single agents tend to become "lost" by the system during the initial cluster forming phase. Once good clusters have formed, these lost agents eventually find their correct cluster, but do so slowly since they are quickly rejected from any incorrect cluster they join. This behavior also occurred with the non-learning agents, although less often.

Unfortunately the "perfect" *shouldSplit* function, shown in Equation 4, requires a-priori knowledge of the inter-cluster distance. On the other hand, the *shouldSplit* function essentially calculates an internal, refined clustering of a cluster. Thus, in theory, *shouldSplit* could be implemented using any existing clustering algorithm, as shown in Equation 5.

```
internal procedure shouldSplit() returns set of Link{
  c1:clustering := ''Clustering in which all members are in a single group'';
  c2:clustering := ''All members clustered into two groups''
  if ( evaluate( c2 ) > evaluate( c1 ) ){ //If c2 is better then c1 according to some measure
    return ''all connections that are between the two groups in c2'';
  else return ∅;
}
```
(5)

In practice, however, the inability to accurately compare clusterings of a data set containing different numbers of clusters makes this difficult. For instance, if we were to use the E^2 measure to implement the *evaluate* function used in Equation 5, E^2_{c2} will always be slightly less then E^2_{c1}, due to the preference of this measure for many clusters. Thus a parameter, γ, needs to be introduced to govern just how much worse E^2_{c2} must be before a cluster is split. There is, however, no value for γ that is appropriate in all cases; as clusters become larger and closer together, including a few incorrect neighboring points in a cluster will make less difference to the E^2 value of the cluster as a whole. In fact, experimentally we were unable to find a value of γ that produced correct clusters even on the straightforward data set in Figure 9. A value large enough to allow clusters to grow also allowed very large clusters to form, which eventually grew to engulf the entire data set. A lower value, on the other hand, split clusters too easily. Thus, more advanced implementations of *shouldSplit* must be explored.

Overall, however, we have succeeded in turning a large distributed clustering problem into many easier small central ones. Since there is no perfect measure of a good clustering, other versions of *shouldSplit* are likely to also require some sort of γ parameter. Nevertheless, γ is far better then the density and cluster size parameters we have used up to now since it allows agents to cluster a much larger range of data sets without adjustment. In the following section we will examine one possible implementation of the *shouldSplit* function, and investigate the range of data sets it can cover.

5 Further Issues and Experimental Analysis

In this section we develop a more advanced variant of the *shouldSplit* function and explore the effect of its decision parameter, γ, on the quality of agent clusterings. We also study the scalability of the resulting clustering algorithm and compare the clusterings it finds to those found by several well-known centralized clustering algorithms.

5.1 Determining Cluster Boundaries

In order to create a suitable implementation of *shouldSplit* we need to reconsider the clustering characteristics stated as a goal in the introduction. Since *shouldSplit* operates on small, known sets of agents there is no need for it to be distributed. There were, however, other reasons to avoid centralization which must still be addressed. These relate to the fact that a central clusterer may need a large amount of information about data points and how to compare them. For this reason, a version of *shouldSplit* that does not require knowledge of the actual data points, or the ways in which they were compared when choosing links, is desirable. Ideally, *shouldSplit* would use only the data that is already collected by the cluster for other steps in the clustering procedure: the length of each of the connecting and matching links. Considering these lengths, we observe that the process or matching and breaking links in the basic agent procedure should create a network of connections within a cluster that somewhat approximates a minimum spanning tree. Thus connecting links between two data clusters are, in most cases, exceptionally long compared to connections that join agents in the same data cluster. These facts can possibly be used to identify unwanted inter-data cluster connections.

In the remaining experiments we use a version of *shouldSplit* that follows this approach. We will consider data clusters to be dense areas of points, surrounded by space with a lower point density. To determine which connections truly belong in an agent cluster, we examine a series containing the cluster's connecting links, ordered by length. For clusters with a relatively consistent density, these connection lengths should be roughly similar. Data clusters with a gradually changing density should result in gradually changing connection lengths. Thus we expect the list of lengths within a cluster to change steadily, while a gap between data subgroups should be indicated by a sudden difference. Such a gap can be detected by estimating the second derivative of the connection length series: $f''(x) \approx (y_2 - 2y_1 + y_0)$, where y_2, y_1 and y_0 are consecutive connection lengths. Sudden changes in connection length will appear as large peaks in this second derivative series. On the other hand, due to the variation of connection lengths in real clusters, even correct clusters will exhibit peaks. To determine what "large" is we calculate the standard deviation of the second derivative series, leaving out the highest and lowest points (since a long connecting link between two data clusters can out shadow all others when clusters are small, decreasing the accuracy): $S_{N-1} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$. The error parameter γ is used to define how many times larger than the standard deviation a peak must be before it is chosen as a splitting

point for the cluster. When such a peak occurs, the corresponding connection is broken. The resulting calculation, given in Equation 6, is illustrated in Figure 10 for a correctly sized and an overly large cluster.

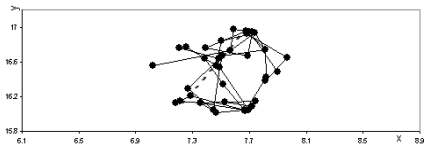
```
internal procedure shouldSplit() returns set of Link{
  lengthSeries: set of distance := ‘‘strength for each connection’’;
  lengthSeries.sortDecreasing(); //Fig 10(c) & 10(d)
  lengthSeries”: set of Real:= estimateSecondDerivate( lengthSeries );//Fig 10(e) & 10(f)
  stdDev: Real:= standardDeviation( lengthSeries” ); //Fig 10(e) dotted line
  //find the unwanted links, actually only returns first one encountered
  breakPosition: Real := ‘‘first member, m, in lengthSeries" for which m ≥ stdDev×γ ’’;
  if breakPosition = null return ∅
  breakLength: Real := ‘‘member of lengthSeries corresponding to breakPosition in lengthSeries"’’;
  return ‘‘any one connection with a strength of breakLength’’ ;
}
```

(6)

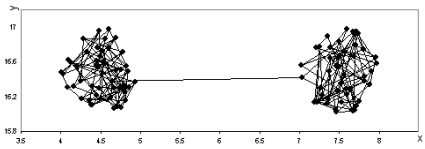
Figure 11(a) shows the resulting clusters using $\gamma = 5$ for a data set containing rows of clusters placed increasingly close together, and with increasingly more points per cluster in each row, going from bottom to the top. There are several factors that influence how easily *shouldSplit* distinguishes clusters. The most important of these is the ratio of the distance separating points within a cluster to the distance between clusters. The bottom row in Figure 11(a) contains clusters with 20 points each. With such sparse clusters small gaps between clusters easily look like part of the cluster. Thus only the most separated cluster, on the far left, is distinguishable to the agents. The next rows up contain clusters of the same area with 40, 80 and 160 points each, respectively. As the cluster density is increased small gaps between clusters become more distinct and it can be seen that the agents become better in separating the clusters from one another.

The *shouldSplit* function described in Equation 6 makes the assumption that the matching/breaking process will keep the number of connections between data clusters to a minimum. For this reason, only the single connection before a peak in the second derivative series is broken when an overgrown cluster is found.[1] This leads to some inaccuracy in separating clusters since often two or three connections can be made between data clusters within a single agent cluster. One possible fix for this is to remove all cycles in the graph formed by the connecting links within each cluster. This limits a cluster to containing only the minimum number of connections to keep it intact. As a result, breaking any connection will break up the cluster. To implement this behavior, each time an internal connection is created, the resulting cycle is identified and its longest connection is broken, leading to the

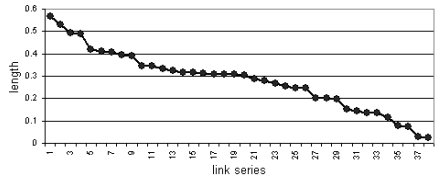
[1] If instead all longer connections were broken agents would have trouble with sparse clusters located next to dense clusters



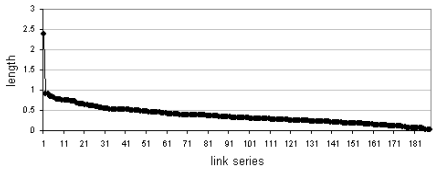
(a) connections in a correct cluster



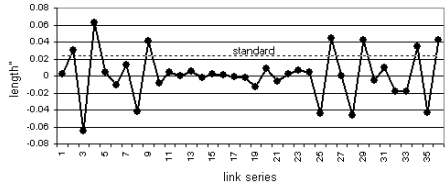
(b) connections in an oversized cluster



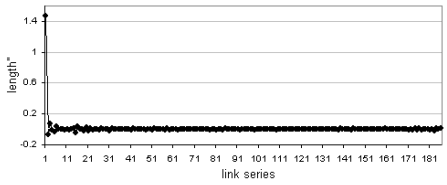
(c) connection length series for the correct cluster



(d) connection length series for the oversized cluster

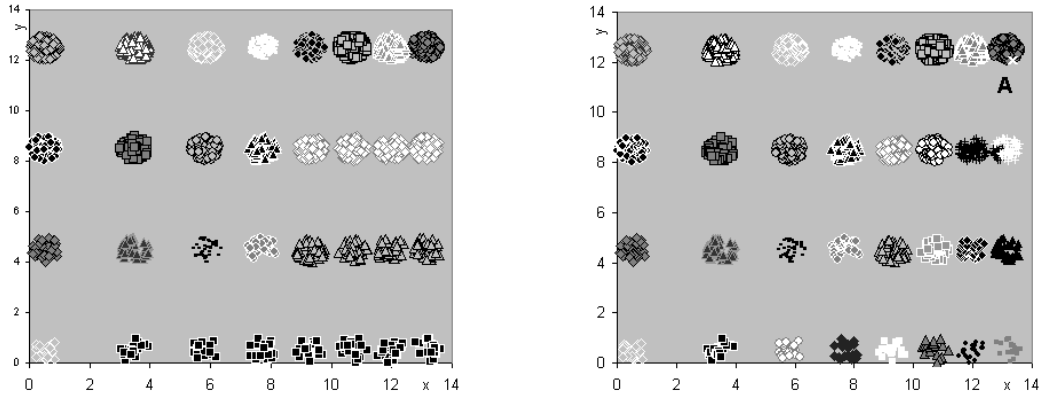


(e) second derivative of 10(c), correct cluster



(f) second derivative of 10(d), oversized cluster

Figure 10: Example *shouldSplit* calculation for a correct and an oversized cluster



(a) All connections kept

(b) Only minimum spanning tree of connections kept

Figure 11: Test of agent's ability to distinguish clusters for varying density/gap-between-clusters ratios.

pseudo code shown in Equation 7.[2]

```
process searchForConnections{
  whenever (''sufficient matches have been found'' & ''no composition changes pending''){
    worstConnection:[ neighbor:Agent, CONNECTING, strength:Strength ] := members.all.acquaintances.min( strength, ( status = CONNECTING ) );
    bestMatch:[ neighbor:Agent, MATCHING, strength:Strength ] := members.all.acquaintances.max( strength, ( status = MATCHING ) );
    if ( members.contains( bestMatch.neighbor ) ){
      if ( bestMatch.strength > worstConnection.strength ){
        ''upgrade bestMatch into a connection'';
        "find the worst connection in the resulting cycle and downgrade it to unmatched";
      }else{
        ''downgrade bestMatch to a nonmatching link'';
        "repeat this whenever clause";
    }else if ( bestMatch.neighbor.c.requestConnection( self ) = ACCEPT ){
      ''upgrade bestMatch into a connection'';
    }else reconsiderComposition();
  }
}
```

(7)

Figure 11(b) shows this new version of the agents, using $\gamma = 7$, run on the data set from Figure 11(a). It can be seen that the agents are now able to distinguish even very small closely packed clusters. Now however, because of the lack of cycles, it is more likely that small parts of the clusters get broken off (and eventually reattached) as the set of found connections improves. This behavior occurs near label A at the small cluster represented by white x's.

[2]Since this results in clusters containing many more internal matches we also need to modify the connection creation behavior to repeatedly make the best matches into connections until it gets to an external connection, rather then just upgrading the single best match.

5.2 Choosing γ

Ideally, the most suitable value of γ would be independent of the number of points in a cluster. Unfortunately, when points are placed randomly within clusters, the second derivative series will always show some amount of random variation. This variation can be large, and the longer the connection series is the more likely we are to see a random fluctuation that results in an incorrect decision to split a cluster. This means that the more agents a cluster contains, the more likely a small γ is to result in incorrect breaks. Thus γ must be set low to accurately distinguish low density clusters that are very close together, but at the same time needs to be kept high enough to avoid accidentally splitting larger clusters. Because of this dichotomy the value of γ is not wholly independent of cluster size. However, because cluster sizes are already limited by coordination costs, the value of γ only needs to cover a sufficiently large range of sizes. We believe a range between 10 and 500 points should be reasonable for many applications.

In order to obtain a more accurate picture of the effect of γ, we examine a series of experiments using a data set containing two data clusters with varying internal densities and a varying separation between them. Depending on the value of γ we observed one of the following behaviors occurring in each trial:

- γ **is set too low**: the clusters are broken into many smaller pieces.
- γ **is set correctly**: the agents easily distinguish the two clusters, though intermittently a few single agents can split off from and later reattach to the cluster edges.
- γ **is set too high**: the clusters are joined into a single large cluster.
- γ **is slightly lower then the correct value**: the clusters are repeatedly found correctly, broken into pieces, then found again.
- γ **is slightly higher then the correct value**: a single cluster is formed, broken up, (often correctly, but also incorrectly) and formed again.

Table 1 summarizes this experiment. The main values in each column are for trials run using the original version of *searchForConnections* (Figure 3). Trials are divided into three categories based on the number and size of clusters found after the trial had been run for a fixed amount of time. The categories are: “correct,” when at least 93.75% of agents were placed in their correct data cluster, “joined,” when at least 93.75% of agents were placed in a single data cluster, and “crumbled” when several smaller data clusters were formed. From this single time point measurement of the cluster sizes it was not possible to distinguish the case where γ was too small from the cases where it was slightly too small or slightly too high. It is interesting to note, however, that if agents could distinguish these cases by observing joining and splitting behavior over time they would have a basis on which to learn a value of γ that fit well with their data.

Table 1 shows the percentage of times a trial resulted in each of the above categories, for a series of 100 trials in each category. We used round clusters with a fixed radius, containing random points generated from an even distribution. For each trial we generated a new data set. There are three variables considered, the value of γ which ranged from 3 to 12, the size (density) of each of the two clusters, which ranged from 8 to 512 agents, and the separation between the two clusters. The ability to distinguish clusters depends on the ratio of the expected distance between nearest neighbors within the cluster and the distance between the two clusters. We consider three values of this ratio, 1:1, 1:4 and 1:12. We expect that for the 1:1 ratio the clusters are so close together that they are unlikely to be distinguishable. The 1:4 ratio produces clusters that are fairly close together and thus relatively difficult to distinguish. The 1:12 ratio produces clusters that should be separable. We also expect a lower γ to more accurately distinguish small clusters while splitting up large clusters, while a higher γ will be less accurate for smaller clusters, but handle large clusters correctly.

The data in Table 1 confirms these expectations. In the 1:1 ratio data sets the clusters were fairly indistinguishable. In the 1:4 ratio data sets the number of joined clusters rises as γ is increased while the number of correctly found clusters first grows but then falls. Furthermore, as cluster size increases correct clusters are found less often and the optimal γ value increases. The 1:12 ratio data sets show that for larger gaps between clusters, clusters of all sizes are usually correctly separated and that a γ of 9 covers our desired size range.

The values in parenthesis in Table 1 give data for the same experiment, run using the version of *searchForConnections* given in Equation 7, which keeps only a minimum spanning tree of connections. Here the trends are not as clearly visible. It can be seen, however, that clusters are more easily distinguished for lower denisity/gap ratios, at a cost of a higher tendency to crumble clusters.

Yet another factor that will change the effectiveness of the *shouldSplit* function is the internal distribution of points within a cluster. The clusters examined so far are generated with a uniform random distribution. The *shouldSplit* function does not prescribe that all cluster connections be approximately the same length. Instead it merely assumes that connection length changes gradually. It should thus also be able to handle other cluster point distributions. To confirm this, Table 1 also shows data for clusters of size 128 containing points generated at random with a Gaussian distribution from the center. To compare with the even distribution clusters we set the radius for these clusters at twice the standard deviation of the point distribution function. We find that this uneven distribution does not cause a problem for the *shouldSplit* function, and that the sparseness of points at the edges of the clusters actually makes them easier to distinguish when the separation is small.

5.3 Scalability

For the form of decentralized clustering studied in this paper there are a fairly large number of factors that affect the speed at which the clusters found by the agents converge to likely data clusters. In particular, as discussed in sections 3 and 4.2.2, coordination is of-

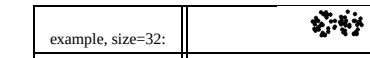
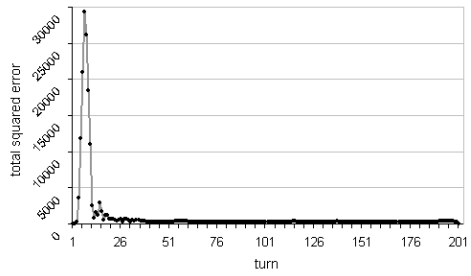
example, size=32:									
separation:	1:1, no gap			1:4, small gap			1:12, larger gap		
	% correct	% crumbled	% joined	% correct	% crumbled	% joined	% correct	% crumbled	% joined
CLUSTER SIZE = 8									
$\gamma = 3$	40 / (63)	3 / (36)	57 / (1)	93 / (78)	4 / (22)	3 / (0)	94 / (78)	6 / (22)	0 / (0)
$\gamma = 6$	13 / (50)	2 / (43)	85 / (7)	78 / (80)	8 / (20)	14 / (0)	89 / (67)	11 / (33)	0 / (0)
$\gamma = 9$	5 / (55)	2 / (29)	93 / (16)	71 / (65)	1 / (34)	28 / (1)	91 / (74)	9 / (26)	0 / (0)
$\gamma = 12$	4 / (55)	6 / (28)	90 / (17)	54 / (65)	1 / (33)	45 / (2)	91 / (80)	9 / (20)	0 / (0)
CLUSTER SIZE = 32									
$\gamma = 3$	19 / (39)	57 / (60)	24 / (1)	56 / (66)	41 / (34)	3 / (0)	63 / (71)	33 / (29)	4 / (0)
$\gamma = 6$	4 / (68)	0 / (19)	96 / (13)	72 / (89)	4 / (9)	24 / (2)	93 / (83)	2 / (17)	5 / (0)
$\gamma = 9$	3 / (56)	0 / (13)	97 / (31)	55 / (91)	1 / (9)	44 / (0)	100 / (95)	0 / (5)	0 / (0)
$\gamma = 12$	1 / (61)	0 / (16)	99 / (23)	38 / (93)	0 / (5)	62 / (2)	100 / (97)	0 / (3)	0 / (0)
CLUSTER SIZE = 128									
$\gamma = 3$	0 / (11)	99 / (88)	1 / (1)	0 / (12)	100 / (88)	0 / (0)	0 / (11)	100 / (89)	0 / (0)
$\gamma = 6$	4 / (48)	2 / (16)	94 / (36)	80 / (100)	0 / (0)	20 / (0)	82 / (100)	2 / (0)	16 / (0)
$\gamma = 9$	0 / (37)	0 / (10)	100 / (53)	42 / (100)	0 / (0)	58 / (0)	92 / (100)	0 / (0)	8 / (0)
$\gamma = 12$	0 / (39)	0 / (8)	100 / (53)	19 / (99)	0 / (0)	81 / (1)	89 / (100)	0 / (0)	11 / (0)
CLUSTER SIZE = 512									
$\gamma = 3$	0 / (0)	100 / (98)	0 / (2)	0 / (2)	100 / (98)	0 / (0)	0 / (0)	100 / (100)	0 / (0)
$\gamma = 6$	3 / (36)	63 / (45)	44 / (19)	29 / (70)	39 / (30)	32 / (0)	55 / (72)	30 / (28)	15 / (0)
$\gamma = 9$	0 / (16)	0 / (5)	100 / (79)	27 / (88)	0 / (1)	73 / (11)	90 / (98)	0 / (1)	10 / (1)
$\gamma = 12$	0 / (5)	0 / (2)	100 / (93)	5 / (69)	0 / (0)	95 / (31)	77 / (89)	0 / (1)	23 / (10)
GAUSSIAN CLUSTERS, CLUSTER SIZE = 128									
$\gamma = 3$	(22)	(76)	(2)	(22)	(78)	(0)	(14)	(86)	(0)
$\gamma = 6$	(96)	(1)	(3)	(97)	(3)	(0)	(100)	(0)	(0)
$\gamma = 9$	(83)	(6)	(11)	(99)	(0)	(1)	(100)	(0)	(0)
$\gamma = 12$	(80)	(1)	(19)	(98)	(0)	(2)	(100)	(0)	(0)

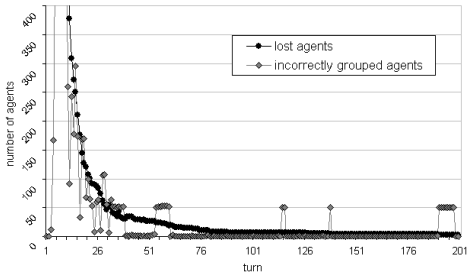
Table 1: Experiments with increasing large pairs of clusters with decreasing gaps between them.

ten achieved by simply waiting until it is likely that a particular state has been reached. The actual length of time chosen can be fairly arbitrary. The exact speed of clustering is also highly dependent on the characteristics of the network the agents communicate over. More important, however, is how the cost of clustering changes as the number of clusters increases. This section presents, for a fixed implementation, a number of experiments to determine how well the algorithm scales with the size of the data set.

In order to measure the relationship between the required time to find a clustering and system size, we consider a series of data sets containing a grid of increasingly many clusters. The size, density and spacing of the clusters is kept constant, thus the experiments measure the change in convergence time as the number of clusters in the data set increases. It should be noted that increasing the number of clusters is only one way of increasing the number of agents. If instead the size of clusters is increased different scalability properties will be seen as the internal coordination cost within clusters grows to outweigh the intra-cluster coordination costs. We preclude this issue by limiting this study to small clusters.

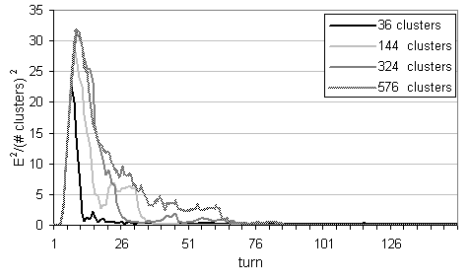


(a) total squared error

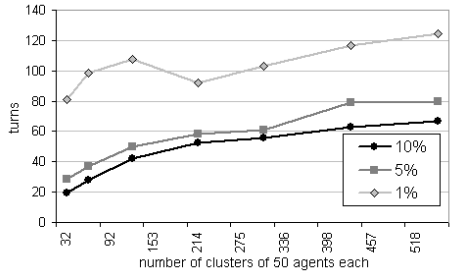


(b) lost and incorrectly grouped agents

Figure 12: Sample run, 36 clusters of 50 point each, agents learning cluster size.



(a) normalized E^2



(b) time to place a given percentage of agents correctly

Figure 13: Comparison of different-sized data sets.

Figure 12 illustrates how agent clusterings converge to underlying data clusters. Figure 12(a) shows the total squared error over time for the smallest data set in the series: the data set from Figure 4. Comparing the E^2 series in Figure 12(a) to that for the simpler agents shown in Figures 6 and 7 shows the effect of allowing clusters to change their maximum size. In Figure 12(a), instead of decreasing smoothly, E^2 fluctuates as agent clusters grow to cover several data clusters and are then split again. This effect can be seen best between turns 11 and 26. Data is shown for a run using the original version of *searchForConnections*, which allows cycles within the connection graph of a cluster (Fig. 3). This version of *searchForConnections* limits the amount of joining and splitting that takes place, making trends clearer. Figure 12(b) was obtained by counting, for each expected data cluster, the number of points it had in common with each agent cluster at a given time. Each data cluster was associated to the agent cluster with which it had the most points in common, and the data cluster points that were not found in their corresponding agent cluster were counted, obtaining the "lost agents" series. This series describes how many agents did not find their

correct cluster, but does not account for agent clusters that cover more then one data cluster. Thus, in the "incorrectly grouped agents" series each agent cluster was associated with the data cluster with which it had the most points in common and points in the agent clusters that did not belong with the corresponding data cluster were counted. Note that if an agent is split off from its cluster it will be counted as a "lost" agent, but if it attaches to another cluster it will also be counted a second time as a "incorrectly grouped" agent. Figure 12(b) shows that the fluctuations in the total squared error in 12(b) are caused by clusters growing too large, incorrectly joining then readjusting to the correct size. For instance at time 116 the value of 50 for the "incorrectly grouped agents" is the same as the underlying data cluster size, indicating that at this point there is one agent cluster that covers two data clusters.

Figure 13 shows how the convergence behavior illustrated in Figure 12 changes as the size of the data sets is increased. Figure 13(a) shows the total squared error over time, for sample runs of data sets with between 36 and 576 clusters. Figure 13(b) was obtained by running 50 trials for each data set in the series, recording the turn at which both the number of "lost agents" and "incorrectly grouped agents" first dropped below 10%, 5% and 1% of the total number of agents. The average of this value over the 50 trials is shown. Figure 13 indicates that the agent procedure scales less then linearly with the number of equally sized clusters in the data set. Centralized clustering algorithms have a processing cost of N or N^2, and require a linear communication cost to gather distributed data. Thus, the agent procedure could compare quite favorably for very large, widely distributed, data sets.

5.4 Quality of Clustering

There are many data set characteristics that can affect the quality of clusterings: the shape of clusters, noise, dimensionality of data, and so forth. For this reason it is difficult to say exactly how good the clusterings found by an algorithm are. In the absence of a standard measure of quality we show a comparison between our agent algorithm and three well-known fundamental clustering algorithms: k-means, the minimal spanning tree version of hierarchical clustering, and a basic implementation of density-based clustering. We consider a data set chosen to have characteristics that show the weaknesses of each.

Figures 14, 15, 16, and 17 show, for each algorithm, example clusterings of this data set. The data set consists of four areas, each intended to illustrate an aspect that can make clustering difficult. Area A has a row of clusters that are connected by increasingly dense bridges of additional points. An algorithm can interpret these bridges as part of the clusters, resulting in neighboring clusters being joined. Area B shows two further data features. First, it contains elongated clusters, which, because of their odd shape, are often split up. Second, the clusters increase in density for left to right, so that the clustering algorithms have to deal with data points that are separated by different distances in each cluster, and with clusters with unequal numbers of data points. Area C extends this concept further. In area B each cluster had twice as many points as the previous one, moving from left to right. Area C accentuates this difference in density. It contains a square of 49 small clusters, containing

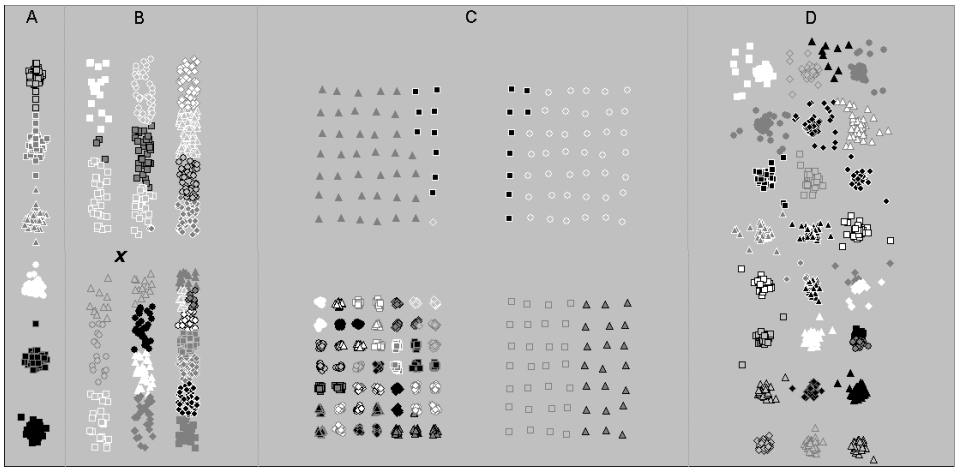


Figure 14: K-means clustering, *k*=88.

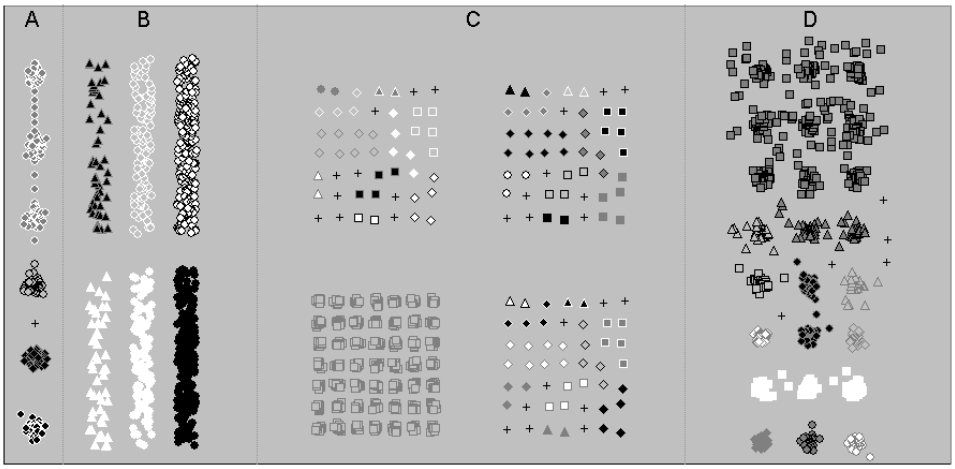


Figure 15: MST clustering, *k*=88. The +'s indicates single points.

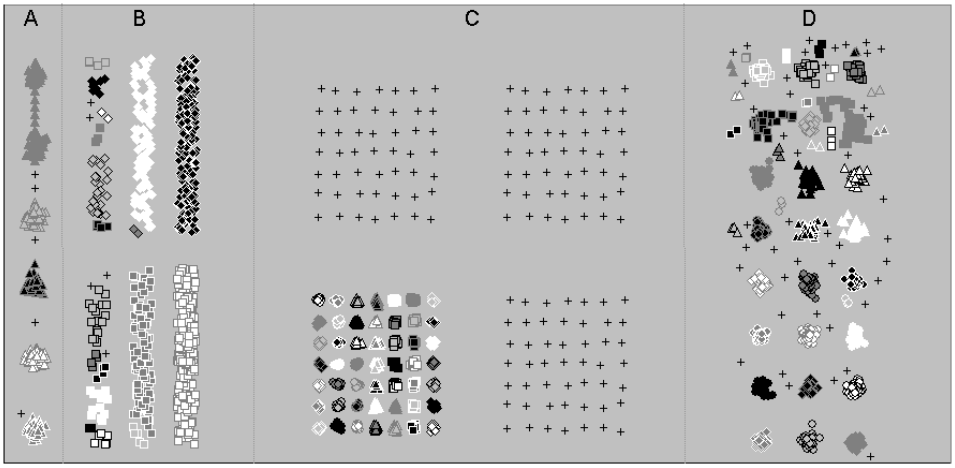


Figure 16: Density-based clustering, d=1.5. The +'s indicate single points.

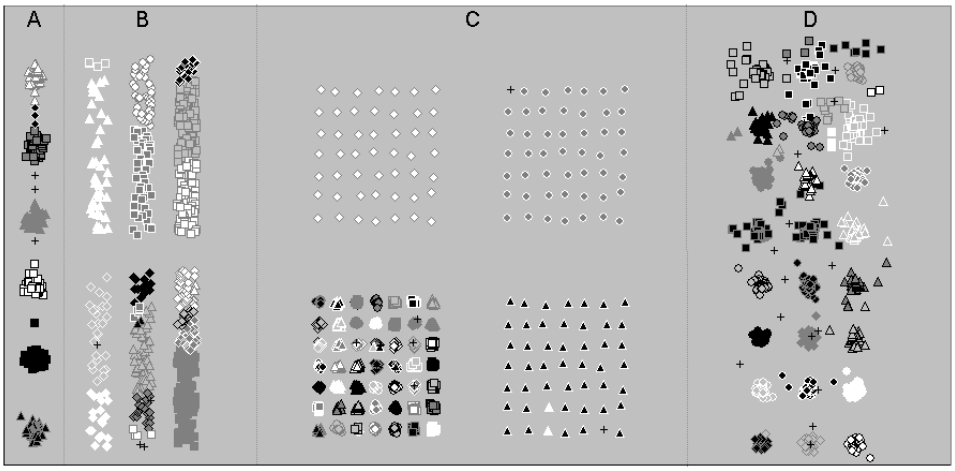


Figure 17: Agent clustering, γ=6. The +'s indicate single points.

20 points each in the bottom left corner, and three sparse clusters produced by translating one point from each of the 49 small clusters. Finally area D illustrates the effect of noise by showing a grid of well separated clusters surrounded by increasingly more random points, from bottom to top. Like bridges, noise points can be interpreted as joining two clusters. To stress this effect the basic round clusters were generated using a Gaussian distribution from the center, in place of the even distribution used in earlier data sets.

Figure 14 shows the clusters found by the k-means algorithm [6], run with $k = 88$, the intended number of clusters in the data set, and random initial centroids. In the k-means algorithm a user picks the number of clusters to be found, k. The algorithm proceeds in rounds. In the first round k points are chosen as "centroids" for clusters, and all the other points are placed in the cluster of their nearest centroid. In the following rounds centroids are shifted to the mid point of each cluster and points are reallocated accordingly. The quality of the clustering is measured by its E^2 value, and rounds are repeated until this measure becomes fixed. By characterizing clusters by a single mid-point, k-means makes the assumption that clusters will have a round shape. Thus it should work well in sections A and D of our data set. It cannot however handle oblong clusters, as can be seen in section B, and even attempts to make round clusters by combining neighboring sections of two oblongs, as at label X. Furthermore, the clustering it finds is dependant on the initial selection of centroids. If two centriods are initially chosen from a single data cluster, that cluster can end up split in the final configuration. Also, since k is fixed, splitting a cluster somewhere in the data set results in two clusters being joined elsewhere. This can lead to problems when there are large differences between clusters in the number of points or density. For example, in section C the large sparse clusters are split while many of the smaller dense clusters are incorrectly joined.

Hierarchical methods are another widely used form of clustering. In hierarchical clustering a series of clusterings is created either by starting with single-point clusters and repeatedly merging the nearest two, or alternatively starting with a single cluster and repeatedly splitting it. The assumption is made that clusters are of roughly equal density, resulting in the globally largest gap between points being an actual space between clusters. In Figure 15 we show the result of a basic form of hierarchical clustering, performed by building the minimum spanning tree between data points and removing its longest edges. We show the clustering containing 88 clusters, the number we know our data set to contain. We see in section B that this method of clustering is much better suited to oblong clusters, since it makes no assumptions about cluster shape. On the other hand, it cannot deal with differing cluster densities. In sections A, C and D it first completely divides the very sparse clusters into single elements before getting around to splitting up dense clusters that are packed together more closely. More advanced versions of hierarchical clustering include heuristics to deal with noise and bridges, however the situation illustrated by area C is fundamentally impossible, for a clustering algorithm based on a global density measure. This point is illustrated by the low quality of clustering shown for area C in Figure 15, where the many dense clusters on the lower left of the area are grouped together while the other three sparse clusters are split into many groups.

Density-based clustering, shown in Figure 16, also assumes that clusters can be parameterized by a characteristic density. This figure illustrates the result of a simple version of density-based clustering in which a fixed global distance of $d = 1.5$ is set as the maximum distance between neigboring points within a cluster. This point of view avoids having to find the correct value for the number of clusters, which is an improvement over the minimum spanning tree algorithm. It still, however, cannot simultaneously distinguish both the dense and sparse clusters in sections B and C. A more complex version of density-based clustering exists, which allows different clusters to have different densities [14]. It assumes that clusters each have a uniform density distribution, and stores a density parameter separately for each cluster. This algorithm takes a step towards the agent approach of describing local characteristics of a cluster, and should be able to handle the clusters we picture in our data set. Though again, we can create a data set with which it will have difficulties by gradually changing the distribution of points within clusters, requiring it to dynamically adapt, just as in our agent-based approach. Density based clustering has the further weakness that it tends to follow bridges between clusters, as has occurred at the top of section A.

Figure 17 shows an example cluster found by our agents, given $\gamma = 6$. Since the agents make the same assumption as the density-based algorithm, namely that clusters are of roughly even densities, we expect it to behave in about the same way. Surprisingly, it manages to cut the dense bridge in section A, but unfortunately also splits up the oblong clusters in section B. This is due to the fact that agents only work with the best connections between points that they have found, not the best ones that exist. Since agents in the middle of a cluster have many more near neighbors than points at the edge of a cluster, or agents in a bridge, they are more likely to find the close connections that hold the cluster together. In dealing with bridges in section A, or with noise in section D this is advantageous and results in better clusterings than expected. On the other hand, it produces problems for attenuated clusters. The main advantage of local adaptation, made possible in agent clustering, can be seen in section C where agents were able to correctly identify both the very sparse and closely packed dense clusters.

6 Discussion of Related Work

Decentralized clustering is an area that has yet to be fully addressed both in the multi-agent systems and in the clustering research communities. To date the multi-agent systems that have been built have either been so small, or have considered homogeneous enough, or simple enough, interactions that problems with centralized directory servers have not yet been encountered. The literature generally takes the approach that very large-scale directory servers will be sufficient. In perhaps the only existing actual large-scale autonomous distributed application, the World Wide Web, this brute force method is used fairly effectively, but with the result that searches are limited to simple keyword matching or fixed category structures, and that search policy can be dictated by a single commercial entity. In a world of truly autonomous agents the inflexibility imposed by a central decision maker

governing matching could nullify many of the proposed benefits promised by multi-agent systems. Full adaptability and support for heterogeneity cannot be achieved if an agent's choice of associates is limited by the contents of, and the matchmaking methods of, a central directory. Scalability is also effected if otherwise independent agents must rely on, and keep updated, a central component.

In the clustering field the emphasis has been on more efficiently handling large data sets using traditional centralized or parallel computing techniques [10], and on exploring algorithms specifically tailored to data sets with particular features [3] [8] [15]. One of the main stumbling blocks the field faces is that data sets can be so varied that, for any algorithm, it is always possible to create an example of a type of data that it cannot handle. Thus it would be possible to continue to research new improved algorithms ad infinitum. This property, however, indicates that clustering is an area for which autonomous agents, with their implicit adaptability to current conditions, would be ideally suited.

Unfortunately the concepts of clustering and decentralization cannot simply be sellotaped together. Centralized clustering is a difficult problem because clusters can have many unknown characteristics, such as size and shape, which make them difficult to define. Decentralized clustering is even more problematic because global information such as the size or range of a data set is unavailable. One of the great difficulties encountered when designing clustering algorithms is defining the term "cluster" in a way that a computer can interpret. By decentralizing clustering we exacerbate this problem.

Intuitively a cluster can be seen as a connected dense region of points, surrounded by a less dense region. Finding clusters is thus finding boundaries between density regions. Because these boundaries are generally fuzzy, computer algorithms need a more concrete definition. To achieve this, additional restrictions are used to make clusters calculable.

The k-means algorithm [6], for instance, fixes the number of clusters in the data set and makes the assumption that clusters are roughly spherical. Given this information, k-means clusters extend to their natural boundaries, provided that centoids are chosen correctly. The k-means passes and various starting heuristics are methods of estimating the "correct" centroids.

Hierarchical algorithms [4] split clusters along the globally largest existing gap to obtain each level. By doing this they in essence say "if there is a density boundary, this edge must cross it." This method identifies boundaries provided that all clusters are of approximately equal density. It leaves aside the problem of choosing which level of the tree contains the correct clustering. By basing this on some statistic, like the total squared error or the number of clusters, additional assumptions are introduced that are, again, dependant on the data set.

Density-based clustering specifically looks for density boundaries by walking through the data set from point to nearest point. Here it is clear to see that a definition of a boundary is required. DBSCAN [2] specifically says that clusters are areas with at least a given density. DBSCLADS [14] improves on this, but must still make the assumption that clusters are of roughly uniform density, with a parameter to define "roughly."

Overall, the standard data set dependent "clues" used by clustering algorithms include the number of clusters (or analogously, the expected cluster size), the minimum gap between

clusters, and the minimum density of clusters. Each of these gives a global standard for the data set, but can be used in decision functions locally. This allows centralized clustering algorithms to be parallelized, provided that global information about the data set can be shared between processing units and that the data set has been sorted so that each processor can contain all the points from a given area [11]. P-CLUSTER [5] for instance parallelized k-means by distributing each k-means pass and synchronizing centroid information between passes.

In this paper we showed that by using agents we could further decentralize the clustering process, allowing us to define different cluster parameters for different areas of the data set, and freeing us from rigid global definitions of similarity. This, however, does not remove the fundamental question: what exactly *is* a cluster? We showed to some extent that this problem can be mitigated by giving agents an ability to learn from their environment, thus eliminating the need to give exact specifications a-priori. We cannot, however, escape it all together: more complex agents could adapt in more directions, but they will always be limited by the methods of adaption bestowed by their designer.

7 Conclusions and Future Work

We have addressed the problem of enabling search in large distributed data sets by organizing data into clusters of similar items. We have argued that moving the decision making required to determine clusters from a central server to distributed agents associated with the data items will improve the scalability of the clustering process. We further contend that this move not only removes the cost of centrally collecting data, but also eliminates the problem of knowing what data to collect when the method of comparison among items is unknown a-priori. By allowing for local decisions on similarity we eliminate the restriction of a global comparison method, and instead make it possible to adapt comparison methods to fit with parameters that can change with location within a data set. This allows for the clustering of an extended range of data sets in which component clusters can have very different characteristics, although it does not eliminate the difficulty of determining the appropriate defining characteristics of clusters for a given application.

This paper proposes an abstract decentralized agent clustering method, based on these observations. It demonstrates, through simulation experiments, that this method can find clusters, showing that the proposed distribution of clustering is actually possible. It further shows, through comparison to clusterings found by other well-known algorithms, that clusterings comparable in quality to those produced by centralized methods can be produced in a decentralized manner. Additionally, an example is given of a method by which agents can locally adapt cluster characteristics within a single data set, reducing the parameter space of the clustering algorithm and thus allowing agents to find certain types of clusters that centralized methods are inherently unable to handle. Finally, data is provided indicating that the scalability properties of the proposed method, in relation to the number of clusters, is indeed an improvement on the, at best, linear cost of centralized clustering.

In future work several additional issues need to be addressed. The experiments presented were simulations on a single machine. Emulations on a distributed network need to be carried out to discover any synchronization subtleties that are not made apparent through simulations. This work considered abstract data sets. It must also be shown that similarity metrics, and the cluster characteristics which agents adapt, can be defined for real applications. Moreover, we have yet to examine the effects of agents having varying similarity metrics. Allowing agents to each choose thier own similarity function increases their autonomy. For applications involving complex data this possibility could be one of the main advantages of performing clustering in a decentralized manner. How well clusters can be found in this situation depends on the relatedness of the different similarity functions to each other. We have done some preliminary work on this subject [10], in which we show that agents clustering text can separately learn key words for their documents. Another possible advantage of decentralized clustering is the ability to easily change clusterings over time as a data set changes. This paper concentrated on clustering data that remained static over time. The model studied, however, views clustering as a continual process. Thus it should be possible for agents to move between clustsers simply by reevaluating their links to neighbors as their data changes.

References

[1] E. Cohen, A. Fiat, and H. Kaplan. Associative search in peer-to-peer networks: Harnessing latent semantics. In *Proceedings of the 22nd INFOCOM Conference*. IEEE Computer Society Press, 2003.

[2] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[3] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. *Information System Journal*, 26(1):35–58, 2001.

[4] A. Jain, M. Murty, and P. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–322, September 1999.

[5] D. Judd, P. McKinley, and A. Jain. Large-scale parallel data clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):871–876, August 1998.

[6] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley and Sons, New York, 1990.

[7] M. Klusch and A. Gerber. Dynamic coalition formation among rational agents. *IEEE Intelligent Systems*, 17(3):42–47, 2002.

[8] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the 20th VLDB Conference*, pages 144–155, 1994.

[9] E. Ogston, B. Overeinder, M. van Steen, and B Brazier. A method for decentralized clustering in large multi-agent systems. In *Proceedings of the Second International Joint Conference on Autonomous Agent and Multi Agent Systems (AAMAS03)*, pages 798–796, Melbourne Australia, 2003.

[10] E. Ogston, M. van Steen, and F. Brazier. Group formation among decentralized autonomous agents. *Applied Artificial Intelligence*, 18(9-10):953–970, October-December 2004.

[11] C. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21:1313–1325, 1995.

[12] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation. *Artif. Intell.*, 101(1-2):165–200, 1998.

[13] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *Proceedings of the 22nd INFOCOM Conference*. IEEE Computer Society Press, 2003.

[14] Xiaowei Xu, Martin Ester, Hans-Peter Kriegel, and Jörg Sander. A distribution-based clustering algorithm for mining in large spatial databases. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pages 324–331, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2):141–182, 1997.